Package ‘readr’

October 1, 2020

Title Read Rectangular Text Data

Version 1.4.0

Description The goal of ‘readr’ is to provide a fast and
friendly way to read rectangular data (like ‘csv’, ‘tsv’, and ‘fwf’).
It is designed to flexibly parse many types of data found in the wild,
while still cleanly failing when data unexpectedly changes.

License GPL (>= 2) | file LICENSE

https://github.com/tidyverse/readr

BugReports https://github.com/tidyverse/readr/issues

Depends R (>= 3.1)

Imports cli,
    clipr,
    crayon,
    hms (>= 0.4.1),
    methods,
    rlang,
    R6,
    tibble,
    utils,
    lifecycle

Suggests covr,
    curl,
    dplyr,
    knitr,
    rmarkdown,
    spelling,
    stringi,
    testthat,
    xml2

LinkingTo BH,
    cpp11

VignetteBuilder knitr

Config/Needs/website pkgdown,
    tidyverse/tidytemplate

Config/testthat/edition 3
Config/testthat/parallel  false
Encoding  UTF-8
Language  en-US
Roxygen  list(markdown = TRUE)
RoxygenNote  7.1.1
SystemRequirements  C++11
RdMacros  lifecycle

R topics documented:
clipboard .................................................. 3
cols .......................................................... 3
cols_condense ............................................... 4
col_skip ..................................................... 5
count_fields ............................................... 5
date_names ................................................ 6
format_delim ............................................... 7
guess_encoding ......................................... 9
locale ....................................................... 9
melt_delim ................................................ 11
melt_fwf .................................................. 14
melt_table ................................................. 16
parse_atomic ............................................. 17
parse_datetime ......................................... 19
parse_factor ............................................. 22
parse_guess ............................................... 23
parse_number ............................................ 24
problems .................................................. 25
readr_example .......................................... 26
read_builtin ............................................. 26
read_delim ................................................. 27
read_file ................................................ 31
read_fwf .................................................. 32
read_lines ............................................... 35
read_log .................................................... 36
read_table ............................................... 38
show_progress ......................................... 41
spec_delim ............................................... 41
type_convert ............................................. 45
write_delim .............................................. 46

Index  50
clipboard

Returns values from the clipboard

**Description**

This is useful in the `read_delim()` functions to read from the clipboard.

**Usage**

```r
clipboard()
```

**See Also**

`read_delim`

---

cols

Create column specification

**Description**

cols() includes all columns in the input data, guessing the column types as the default. `cols_only()` includes only the columns you explicitly specify, skipping the rest.

**Usage**

```r
cols(..., .default = col_guess())
cols_only(...)
```

**Arguments**

- `...`: Either column objects created by `col_*()`, or their abbreviated character names (as described in the `col_types` argument of `read_delim()`). If you're only overriding a few columns, it's best to refer to columns by name. If not named, the column types must match the column names exactly.
- `.default`: Any named columns not explicitly overridden in `...` will be read with this column type.

**Details**

The available specifications are: (with string abbreviations in brackets)

- `col_logical()` [l], containing only T, F, TRUE or FALSE.
- `col_integer()` [i], integers.
- `col_double()` [d], doubles.
- `col_character()` [c], everything else.
- `col_factor(levels, ordered)` [f], a fixed set of values.
- `col_date(format = "")` [D]: with the locale's date_format.
cols_condense

- col_time(format = "") [t]: with the locale’s time_format.
- col_datetime(format = "") [T]: ISO8601 date times
- col_number() [n], numbers containing the grouping_mark
- col_skip() [_, -], don’t import this column.
- col_guess() [?], parse using the "best" type based on the input.

See Also

Other parsers: col_skip(), cols_condense(), parse_datetime(), parse_factor(), parse_guess(), parse_logical(), parse_number(), parse_vector()
**col_skip**

Value

A col_spec object.

See Also

Other parsers: `col_skip()`, `cols()`, `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_number()`, `parse_vector()`

Examples

```r
def <- read_csv(readr_example("mtcars.csv"))
s <- spec(df)
s
cols_condense(s)
```

---

col_skip 

**Skip a column**

Description

Use this function to ignore a column when reading in a file. To skip all columns not otherwise specified, use `cols_only()`.

Usage

col_skip()

See Also

Other parsers: `cols_condense()`, `cols()`, `parse_datetime()`, `parse_factor()`, `parse_guess()`, `parse_logical()`, `parse_number()`, `parse_vector()`

---

count_fields  

**Count the number of fields in each line of a file**

Description

This is useful for diagnosing problems with functions that fail to parse correctly.

Usage

count_fields(file, tokenizer, skip = 0, n_max = -1L)
Arguments

file
Either a path to a file, a connection, or literal data (either a single string or a raw vector).
Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed.
Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.
Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path) or be a vector of greater than length 1.
Using a value of \texttt{clipboard()} will read from the system clipboard.

tokenizer
A tokenizer that specifies how to break the file up into fields, e.g., \texttt{tokenizer_csv()}, \texttt{tokenizer_fwf()}

skip
Number of lines to skip before reading data.

n_max
Optionally, maximum number of rows to count fields for.

Examples

\begin{verbatim}
\texttt{count_fields(readr_example("mtcars.csv"), tokenizer_csv())}
\end{verbatim}

\section*{date_names}

\textit{Create or retrieve date names}

Description

When parsing dates, you often need to know how weekdays of the week and months are represented as text. This pair of functions allows you to either create your own, or retrieve from a standard list. The standard list is derived from ICU (http://site.icu-project.org) via the stringi package.

Usage

\begin{verbatim}
date_names(mon, mon_ab = mon, day, day_ab = day, am_pm = c("AM", "PM"))
date_names_lang(language)
date_names_langs()
\end{verbatim}

Arguments

mon, mon_ab
Full and abbreviated month names.

day, day_ab
Full and abbreviated week day names. Starts with Sunday.

am_pm
Names used for AM and PM.

language
A BCP 47 locale, made up of a language and a region, e.g. "en_US" for American English. See date_names_langs() for a complete list of available locales.

Examples

\begin{verbatim}
date_names_lang("en")
date_names_lang("ko")
date_names_lang("fr")
\end{verbatim}
**format_delim**

*Convert a data frame to a delimited string*

**Description**

These functions are equivalent to `write_csv()` etc., but instead of writing to disk, they return a string.

**Usage**

```r
format_delim(
  x,
  delim,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote_escape = "double",
  eol = "\n"
)

format_csv(
  x,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote_escape = "double",
  eol = "\n"
)

format_csv2(
  x,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote_escape = "double",
  eol = "\n"
)

format_tsv(
  x,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote_escape = "double",
  eol = "\n"
)
```

**Arguments**

- **x**: A data frame.
format_delim

delim  Delimiter used to separate values. Defaults to " " for write_delim(), "," for write_excel_csv() and ";" for write_excel_csv2(). Must be a single character.

na  String used for missing values. Defaults to NA. Missing values will never be quoted; strings with the same value as na will always be quoted.

append  If FALSE, will overwrite existing file. If TRUE, will append to existing file. In both cases, if the file does not exist a new file is created.

col_names  If FALSE, column names will not be included at the top of the file. If TRUE, column names will be included. If not specified, col_names will take the opposite value given to append.

quote_escape  The type of escaping to use for quoted values, one of "double", "backslash" or "none". You can also use FALSE, which is equivalent to "none". The default is "double", which is expected format for Excel.

eol  The end of line character to use. Most commonly either "\n" for Unix style newlines, or "\r\n" for Windows style newlines.

Value

A string.

Output

Factors are coerced to character. Doubles are formatted to a decimal string using the grisu3 algorithm. POSIXct values are formatted as ISO8601 with a UTC timezone Note: POSIXct objects in local or non-UTC timezones will be converted to UTC time before writing.

All columns are encoded as UTF-8. write_excel_csv() and write_excel_csv2() also include a UTF-8 Byte order mark which indicates to Excel the csv is UTF-8 encoded.

write_excel_csv2() and write_csv2 were created to allow users with different locale settings to save .csv files using their default settings (e.g. ; as the column separator and , as the decimal separator). This is common in some European countries.

Values are only quoted if they contain a comma, quote or newline.

The write_*() functions will automatically compress outputs if an appropriate extension is given. Three extensions are currently supported: .gz for gzip compression, .bz2 for bzip2 compression and .xz for lzma compression. See the examples for more information.

References


Examples

data(band_members, package = "dplyr")
# format_()* functions are useful for testing and reprexes
cat(format_csv(band_members))
cat(format_tsv(band_members))
cat(format_delim(band_members, ";"))

# Specifying missing values
df <- data.frame(x = c(1, NA, 3))
format_csv(df, na = "missing")
# Quotes are automatically added as needed
```r
def <- data.frame(x = c("a ", ",", ",", 
))
cat(format_csv(df))
```

---

### guess_encoding

**Guess encoding of file**

**Description**

Uses `stringi::stri_enc_detect()`: see the documentation there for caveats.

**Usage**

```r
guess_encoding(file, n_max = 10000, threshold = 0.2)
```

**Arguments**

- **file**: A character string specifying an input as specified in `datasource()`, a raw vector, or a list of raw vectors.
- **n_max**: Number of lines to read. If n_max is -1, all lines in file will be read.
- **threshold**: Only report guesses above this threshold of certainty.

**Value**

A tibble

**Examples**

```r
guess_encoding(readr_example("mtcars.csv"))
guess_encoding(read_lines_raw(readr_example("mtcars.csv")))
guess_encoding(read_file_raw(readr_example("mtcars.csv")))
guess_encoding("a\n\u00b5\u00b5")
```

---

### locale

**Create locales**

**Description**

A locale object tries to capture all the defaults that can vary between countries. You set the locale in once, and the details are automatically passed on down to the columns parsers. The defaults have been chosen to match R (i.e. US English) as closely as possible. See vignette("locales") for more details.
Usage
locale(
  date_names = "en",
  date_format = "%AD",
  time_format = "%AT",
  decimal_mark = ".",
  grouping_mark = ",",
  tz = "UTC",
  encoding = "UTF-8",
  asciify = FALSE
)

default_locale()

Arguments

date_names  Character representations of day and month names. Either the language code as string (passed on to date_names_lang()) or an object created by date_names().
date_format, time_format  Default date and time formats.
decimal_mark, grouping_mark  Symbols used to indicate the decimal place, and to chunk larger numbers. Decimal mark can only be . or ..
tz  Default tz. This is used both for input (if the time zone isn’t present in individual strings), and for output (to control the default display). The default is to use "UTC", a time zone that does not use daylight savings time (DST) and hence is typically most useful for data. The absence of time zones makes it approximately 50x faster to generate UTC times than any other time zone. Use "" to use the system default time zone, but beware that this will not be reproducible across systems.

For a complete list of possible time zones, see OlsonNames(). Americans, note that "EST" is a Canadian time zone that does not have DST. It is not Eastern Standard Time. It’s better to use "US/Eastern", "US/Central" etc.

encoding  Default encoding. This only affects how the file is read - readr always converts the output to UTF-8.

asciify  Should diacritics be stripped from date names and converted to ASCII? This is useful if you’re dealing with ASCII data where the correct spellings have been lost. Requires the stringi package.

Examples
locale()
locale("fr")

# South American locale
locale("es", decimal_mark = ",")
melt_delim

**Return melted data for each token in a delimited file (including csv & tsv)**

**Description**

For certain non-rectangular data formats, it can be useful to parse the data into a melted format where each row represents a single token.

**Usage**

```r
melt_delim(
  file,
  delim,
  quote = "\\",
  escape_backslash = FALSE,
  escape_double = TRUE,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  comment = "",
  trim_ws = FALSE,
  skip = 0,
  n_max = Inf,
  progress = show_progress(),
  skip_empty_rows = FALSE
)
```

```r
melt_csv(
  file,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\\",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  progress = show_progress(),
  skip_empty_rows = FALSE
)
```

```r
melt_csv2(
  file,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\\",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
)```
melt_delim

n_max = Inf,
progress = show_progress(),
skip_empty_rows = FALSE
)
melt_tsv(
file,
locale = default_locale(),
na = c("", "NA"),
quoted_na = TRUE,
quote = "\"",
comment = "",
trim_ws = TRUE,
skip = 0,
n_max = Inf,
progress = show_progress(),
skip_empty_rows = FALSE
)

Arguments

file Either a path to a file, a connection, or literal data (either a single string or a raw vector).

Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.

Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path) or be a vector of greater than length 1.

Using a value of clipboard() will read from the system clipboard.

delim Single character used to separate fields within a record.

quote Single character used to quote strings.

escape_backslash Does the file use backslashes to escape special characters? This is more general than escape_double as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like \
.

escape_double Does the file escape quotes by doubling them? i.e. If this option is TRUE, the value """" represents a single quote, ".

locale The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

na Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values.

quoted_na Should missing values inside quotes be treated as missing values (the default) or strings.

comment A string used to identify comments. Any text after the comment characters will be silently ignored.
melt_delim

trim_ws  Should leading and trailing whitespace be trimmed from each field before parsing it?
skip     Number of lines to skip before reading data.
n_max    Maximum number of records to read.
progress Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The display is updated every 50,000 values and will only display if estimated reading time is 5 seconds or more. The automatic progress bar can be disabled by setting option readr.show_progress to FALSE.
skip_empty_rows Should blank rows be ignored altogether? i.e. If this option is TRUE then blank rows will not be represented at all. If it is FALSE then they will be represented by NA values in all the columns.

Details

melt_csv() and melt_tsv() are special cases of the general melt_delim(). They’re useful for reading the most common types of flat file data, comma separated values and tab separated values, respectively. melt_csv2() uses ; for the field separator and , for the decimal point. This is common in some European countries.

Value

A tibble() of four columns:

- **row**, the row that the token comes from in the original file
- **col**, the column that the token comes from in the original file
- **data_type**, the data type of the token, e.g. "integer", "character", "date", guessed in a similar way to the guess_parser() function.
- **value**, the token itself as a character string, unchanged from its representation in the original file.

If there are parsing problems, a warning tells you how many, and you can retrieve the details with problems().

See Also

read_delim() for the conventional way to read rectangular data from delimited files.

Examples

```r
# Input sources -------------------------------------------------------------
# Read from a path
melt_csv(readr_example("mtcars.csv"))
melt_csv(readr_example("mtcars.csv.zip"))
melt_csv(readr_example("mtcars.csv.bz2"))
## Not run:
melt_csv("https://github.com/tidyverse/readr/raw/master/inst/extdata/mtcars.csv")
## End(Not run)

# Or directly from a string (must contain a newline)
melt_csv("x,y
1,2
3,4")
```
To import empty cells as 'empty' rather than 'NA'
melt_csv("x,y\nNA,\"\", na = "NA")

# File types
melt_csv("a,b\n1.0,2.0")
melt_csv2("a;b\n1,0;2,0")
melt_tsv("a\tb\n1.0\t2.0")
melt_delim("a|b\n1.0|2.0", delim = ")

---

**melt_fwf**

Return melted data for each token in a fixed width file

---

**Description**

For certain non-rectangular data formats, it can be useful to parse the data into a melted format where each row represents a single token.

**Usage**

```r
melt_fwf(
  file,
  col_positions,
  locale = default_locale(),
  na = c("","NA"),
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  progress = show_progress(),
  skip_empty_rows = FALSE
)
```

**Arguments**

- **file** Either a path to a file, a connection, or literal data (either a single string or a raw vector).
  - Files ending in `.gz`, `.bz2`, `.xz`, or `.zip` will be automatically uncompressed.
  - Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.
  - Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path) or be a vector of greater than length 1.
  - Using a value of `clipboard()` will read from the system clipboard.

- **col_positions** Column positions, as created by `fwf_empty()`, `fwf_widths()` or `fwf_positions()`. To read in only selected fields, use `fwf_positions()`. If the width of the last column is variable (a ragged fwf file), supply the last end position as NA.

- **locale** The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
melt_fwf

Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values.

A string used to identify comments. Any text after the comment characters will be silently ignored.

Should leading and trailing whitespace be trimmed from each field before parsing it?

Number of lines to skip before reading data.

Maximum number of records to read.

Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The display is updated every 50,000 values and will only display if estimated reading time is 5 seconds or more. The automatic progress bar can be disabled by setting option readr.show_progress to FALSE.

Should blank rows be ignored altogether? i.e. If this option is TRUE then blank rows will not be represented at all. If it is FALSE then they will be represented by NA values in all the columns.

melt_fwf() parses each token of a fixed width file into a single row, but it still requires that each field is in the same in every row of the source file.

See Also

melt_table() to melt fixed width files where each column is separated by whitespace, and read_fwf() for the conventional way to read rectangular data from fixed width files.

Examples

fwf_sample <- readr_example("fwf-sample.txt")
cat(read_lines(fwf_sample))

# You can specify column positions in several ways:
# 1. Guess based on position of empty columns
melt_fwf(fwf_sample, fwf_empty(fwf_sample, col_names = c("first", "last", "state", "ssn")))
# 2. A vector of field widths
melt_fwf(fwf_sample, fwf_widths(c(20, 10, 12), c("name", "state", "ssn")))
# 3. Paired vectors of start and end positions
melt_fwf(fwf_sample, fwf_positions(c(1, 30), c(10, 42), c("name", "ssn")))
# 4. Named arguments with start and end positions
melt_fwf(fwf_sample, fwf_cols(name = c(1, 10), ssn = c(30, 42)))
# 5. Named arguments with column widths
melt_fwf(fwf_sample, fwf_cols(name = 20, state = 10, ssn = 12))
melt_table

Return melted data for each token in a whitespace-separated file

Description

For certain non-rectangular data formats, it can be useful to parse the data into a melted format where each row represents a single token.

melt_table() and melt_table2() are designed to read the type of textual data where each column is separated by one (or more) columns of space. melt_table2() allows any number of whitespace characters between columns, and the lines can be of different lengths.
melt_table() is more strict, each line must be the same length, and each field is in the same position in every line. It first finds empty columns and then parses like a fixed width file.

Usage

melt_table(
  file,
  locale = default_locale(),
  na = "NA",
  skip = 0,
  n_max = Inf,
  guess_max = min(n_max, 1000),
  progress = show_progress(),
  comment = "",
  skip_empty_rows = FALSE
)

melt_table2(
  file,
  locale = default_locale(),
  na = "NA",
  skip = 0,
  n_max = Inf,
  progress = show_progress(),
  comment = "",
  skip_empty_rows = FALSE
)

Arguments

file

Either a path to a file, a connection, or literal data (either a single string or a raw vector).
Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed.
Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.
Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path) or be a vector of greater than length 1.
Using a value of `clipboard()` will read from the system clipboard.

The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.

Number of lines to skip before reading data.

Maximum number of lines to read.

Maximum number of records to use for guessing column types.

Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The display is updated every 50,000 values and will only display if estimated reading time is 5 seconds or more. The automatic progress bar can be disabled by setting option `readr.show_progress` to `FALSE`.

A string used to identify comments. Any text after the comment characters will be silently ignored.

Should blank rows be ignored altogether? i.e. If this option is `TRUE` then blank rows will not be represented at all. If it is `FALSE` then they will be represented by `NA` values in all the columns.

See Also

`melt_fwf()` to melt fixed width files where each column is not separated by whitespace. `melt_fwf()` is also useful for reading tabular data with non-standard formatting. `read_table()` is the conventional way to read tabular data from whitespace-separated files.

Examples

```r
# One corner from http://www.masseyratings.com/cf/compare.htm
massey <- readr_example("massey-rating.txt")
cat(read_file(massey))
melt_table(massey)

# Sample of 1978 fuel economy data from
# http://www.fueleconomy.gov/feg/epadata/78data.zip
epa <- readr_example("epa78.txt")
cat(read_file(epa))
melt_table(epa)
```

Description

Use `parse_*()` if you have a character vector you want to parse. Use `col_*()` in conjunction with a `read_*()` function to parse the values as they're read in.
Usage

parse_logical(x, na = c("", "NA"), locale = default_locale(), trim_ws = TRUE)
parse_integer(x, na = c("", "NA"), locale = default_locale(), trim_ws = TRUE)
parse_double(x, na = c("", "NA"), locale = default_locale(), trim_ws = TRUE)
parse_character(x, na = c("", "NA"), locale = default_locale(), trim_ws = TRUE)
col_logical()
col_integer()
col_double()
col_character()

Arguments

x Character vector of values to parse.
na Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values.
locale The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
trim_ws Should leading and trailing whitespace be trimmed from each field before parsing it?

See Also

Other parsers: col_skip(), cols_condense(), cols(), parse_datetime(), parse_factor(), parse_guess(), parse_number(), parse_vector()

Examples

parse_integer(c("1", "2", "3"))
parse_double(c("1", "2", "3.123"))
parse_number("$1,123,456.00")

# Use locale to override default decimal and grouping marks
es_MX <- locale("es", decimal_mark = ",")
parse_number("$1.123.456,00", locale = es_MX)

# Invalid values are replaced with missing values with a warning.
x <- c("1", "2", "3", "-")
parse_double(x)
# Or flag values as missing
parse_double(x, na = "-")
Description

Parse date/times

Usage

```r
parse_datetime(
  x,
  format = "",
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE
)
```

```r
parse_date(
  x,
  format = "",
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE
)
```

```r
parse_time(
  x,
  format = "",
  na = c("", "NA"),
  locale = default_locale(),
  trim_ws = TRUE
)
```

```r
col_datetime(format = "")
col_date(format = "")
col_time(format = "")
```

Arguments

- **x**: A character vector of dates to parse.
- **format**: A format specification, as described below. If set to "", date times are parsed as ISO8601, dates and times used the date and time formats specified in the `locale()`.
  Unlike `strptime()`, the format specification must match the complete string.
- **na**: Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
- **locale**: The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that

controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

trim_ws
Should leading and trailing whitespace be trimmed from each field before parsing it?

Value
A POSIXct() vector with tzone attribute set to tz. Elements that could not be parsed (or did not generate valid dates) will be set to NA, and a warning message will inform you of the total number of failures.

Format specification
readr uses a format specification similar to strptime(). There are three types of element:

1. Date components are specified with "\. For example "\ month and "\ (i.e. Jan 1st) if not present, for example if only a year is given.
2. Whitespace is any sequence of zero or more whitespace characters.
3. Any other character is matched exactly.

parse_datetime() recognises the following format specifications:

- Month: "\ locale), "\ (e.g. "Jan 1st) if not present.
- Day: "\( abbreviated name in current locale).
- Hour: "\ use h (and not H) if your times represent durations longer than one day.
- Minutes: "\ (and not M)
- Seconds: "\ (and not S)
- Time zone: "\ offset from UTC, e.g. "+0800")
- AM/PM indicator: "\ (and not AM/PM)
- Non-digits: "\ "
- Automatic parsers: "\ "
- Shortcuts: "\ "

ISO8601 support
Currently, readr does not support all of ISO8601. Missing features:

- Week & weekday specifications, e.g. "2013-W05", "2013-W05-10"
- Ordinal dates, e.g. "2013-095".
- Using commas instead of a period for decimal separator

The parser is also a little laxer than ISO8601:

- Dates and times can be separated with a space, not just T.
- Mostly correct specifications like "2009-05-19 14:" and "200912-01" work.

See Also
Other parsers: col_skip(), cols_condense(), cols(), parse_factor(), parse_guess(), parse_logical(), parse_number(), parse_vector()
Examples

```r
# Format strings --------------------------------------------------------
parse_datetime("01/02/2010", "%d/%m/%Y")
parse_datetime("01/02/2010", "%m/%d/%Y")
# Handle any separator
parse_datetime("01/02/2010", "%m%.%d%.%Y")

# Dates look the same, but internally they use the number of days since
# 1970-01-01 instead of the number of seconds. This avoids a whole lot
# of troubles related to time zones, so use if you can.
parse_date("01/02/2010", "%d/%m/%Y")
parse_date("01/02/2010", "%m/%d/%Y")

# You can parse timezones from strings (as listed in OlsonNames())
parse_datetime("2010/01/01 12:00 US/Central", "%Y/%m/%d %H:%M %Z")
# Or from offsets
parse_datetime("2010/01/01 12:00 -0600", "%Y/%m/%d %H:%M %z")

# Use the locale parameter to control the default time zone
# (but note UTC is considerably faster than other options)
locale = locale(tz = "US/Central")
parse_datetime("2010/01/01 12:00", "%Y/%m/%d %H:%M", locale)
parse_datetime("2010/01/01 12:00", "%Y/%m/%d %H:%M", locale = locale(tz = "US/Eastern"))

# Unlike strftime, the format specification must match the complete
# string (ignoring leading and trailing whitespace). This avoids common
# errors:
strptime("01/02/2010", "%d/%m/%y")
parse_datetime("01/02/2010", "%d/%m/%y")

# Failures -------------------------------------------------------------
parse_datetime("01/01/2010", "%d/%m/%Y")
parse_datetime(c("01/ab/2010", "32/01/2010"), "%d/%m/%Y")

# Locales --------------------------------------------------------------
# By default, readr expects English date/times, but that's easy to change'
parse_datetime("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
parse_datetime("1 enero 2015", "%d %B %Y", locale = locale("es"))

# ISO8601 --------------------------------------------------------------
# With separators
parse_datetime("1979-10-14")
parse_datetime("1979-10-14T10")
parse_datetime("1979-10-14T10:11")
parse_datetime("1979-10-14T10:11:12")
parse_datetime("1979-10-14T10:11:12.12345")

# Without separators
parse_datetime("19791014")
parse_datetime("19791014T101112")

# Time zones
us_central <- locale(tz = "US/Central")
parse_datetime("1979-10-14T10", locale = us_central)
parse_datetime("1979-10-14T10:0500", locale = us_central)
```
parse_factor

Description

parse_factor is similar to factor(), but will generate warnings if elements of x are not found in levels.

Usage

parse_factor(
  x,  
  levels = NULL,  
  ordered = FALSE,  
  na = c("", "NA"),  
  locale = default_locale(),  
  include_na = TRUE,  
  trim_ws = TRUE
)

col_factor(levels = NULL, ordered = FALSE, include_na = FALSE)

Arguments

x Character vector of values to parse.
levels Character vector providing set of allowed levels. If NULL, will generate levels based on the unique values of x, ordered by order of appearance in x.
ordered Is it an ordered factor?
na Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values.
locale The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
include_na If NA are present, include as an explicit factor to level?
trim_ws Should leading and trailing whitespace be trimmed from each field before parsing it?

See Also

Other parsers: col_skip(), cols_condense(), cols(), parse_datetime(), parse_guess(), parse_logical(), parse_number(), parse_vector()
Examples

```r
c parse_factor(c("a", "b"), letters)
x <- c("cat", "dog", "caw")
levels <- c("cat", "dog", "cow")

# Base R factor() silently converts unknown levels to NA
x1 <- factor(x, levels)

# parse_factor generates a warning & problems
x2 <- parse_factor(x, levels)

# Using an argument of `NULL` will generate levels based on values of `x`
x2 <- parse_factor(x, levels = NULL)
```

---

### Description

`parse_guess()` returns the parser vector; `guess_parser()` returns the name of the parser. These functions use a number of heuristics to determine which type of vector is "best". Generally they try to err of the side of safety, as it's straightforward to override the parsing choice if needed.

### Usage

```r
parse_guess(
x,
na = c("", "NA"),
locale = default_locale(),
trim_ws = TRUE,
guess_integer = FALSE
)

col_guess()

guess_parser(
x,
locale = default_locale(),
guess_integer = FALSE,
na = c("", "NA")
)
```

### Arguments

- **x**: Character vector of values to parse.
- **na**: Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
- **locale**: The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
#### Description

This drops any non-numeric characters before or after the first number. The grouping mark specified by the locale is ignored inside the number.

#### Usage

```r
col_number()
```

#### Arguments

- **x**
  - Character vector of values to parse.
- **na**
  - Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
- **locale**
  - The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.
- **trim_ws**
  - Should leading and trailing whitespace be trimmed from each field before parsing it?
problems

Value
A numeric vector (double) of parsed numbers.

See Also
Other parsers: col_skip(), cols_condense(), cols(), parse_datetime(), parse_factor(), parse_guess(), parse_logical(), parse_vector()

Examples
## These all return 1000
parse_number("$1,000")  ## leading $ and grouping character , ignored
parse_number("euro1,000")  ## leading non-numeric euro ignored
parse_number("1,234.56")
## explicit locale specifying European grouping and decimal marks
parse_number("1.234,56", locale = locale(decimal_mark = ",", grouping_mark = "."))
## SI/ISO 31-0 standard spaces for number grouping
parse_number("1 234.56", locale = locale(decimal_mark = ",", grouping_mark = " "))
## Specifying strings for NAs
parse_number(c("1", "2", "3", "NA"))
parse_number(c("1", "2", "3", "NA", "Nothing"), na = c("NA", "Nothing"))

problems

Retrieve parsing problems

Description
Readr functions will only throw an error if parsing fails in an unrecoverable way. However, there are lots of potential problems that you might want to know about - these are stored in the problems attribute of the output, which you can easily access with this function. stop_for_problems() will throw an error if there are any parsing problems: this is useful for automated scripts where you want to throw an error as soon as you encounter a problem.

Usage
problems(x)
stop_for_problems(x)

Arguments
x An data frame (from read_*()) or a vector (from parse_*()).

Value
A data frame with one row for each problem and four columns:
row,col Row and column of problem
expected What readr expected to find
actual What it actually got
Examples

```r
x <- parse_integer(c("1X", "blah", "3"))
problems(x)

y <- parse_integer(c("1", "2", "3"))
problems(y)
```

readr_example

Get path to readr example

Description

readr comes bundled with a number of sample files in its inst/extdata directory. This function make them easy to access

Usage

```r
readr_example(file = NULL)
```

Arguments

- `file` Name of file. If NULL, the example files will be listed.

Examples

```
readr_example()
readr_example("challenge.csv")
```

read_builtin

Read built-in object from package

Description

Consistent wrapper around `data()` that forces the promise. This is also a stronger parallel to loading data from a file.

Usage

```r
read_builtin(x, package = NULL)
```

Arguments

- `x` Name (character string) of data set to read.
- `package` Name of package from which to find data set. By default, all attached packages are searched and then the 'data' subdirectory (if present) of the current working directory.

Value

An object of the built-in class of `x`. 
read_delim

**Examples**

```r
if (requireNamespace("dplyr")) {
  read_builtin("starwars", "dplyr")
  read_builtin("storms", "dplyr")
}
```

**Description**

`read_csv()` and `read_tsv()` are special cases of the general `read_delim()`. They're useful for reading the most common types of flat file data, comma separated values and tab separated values, respectively. `read_csv2()` uses ; for the field separator and , for the decimal point. This is common in some European countries.

**Usage**

```r
read_delim(
  file,
  delim,
  quote = "\"",
  escape_backslash = FALSE,
  escape_double = TRUE,
  col_names = TRUE,
  col_types = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  comment = "",
  trim_ws = FALSE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  progress = show_progress(),
  skip_empty_rows = TRUE
)
```

```r
read_csv(
  file,
  col_names = TRUE,
  col_types = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\"",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
)```
guess_max = min(1000, n_max),
progress = show_progress(),
skip_empty_rows = TRUE
)

read_csv2(
  file,
  col_names = TRUE,
  col_types = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\\",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  progress = show_progress(),
  skip_empty_rows = TRUE
)

read_tsv(
  file,
  col_names = TRUE,
  col_types = NULL,
  locale = default_locale(),
  na = c("", "NA"),
  quoted_na = TRUE,
  quote = "\\",
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  guess_max = min(1000, n_max),
  progress = show_progress(),
  skip_empty_rows = TRUE
)

Arguments

file               Either a path to a file, a connection, or literal data (either a single string or a raw vector).
                   Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed.
                   Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.
                   Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path) or be a vector of greater than length 1.
                   Using a value of clipboard() will read from the system clipboard.

delim               Single character used to separate fields within a record.
quote
- Single character used to quote strings.

escape_backslash
- Does the file use backslashes to escape special characters? This is more general than escape_double as backslashes can be used to escape the delimiter character, the quote character, or to add special characters like \n.

escape_double
- Does the file escape quotes by doubling them? i.e. If this option is TRUE, the value """" represents a single quote, "".

col_names
- Either TRUE, FALSE or a character vector of column names.
  - If TRUE, the first row of the input will be used as the column names, and will not be included in the data frame. If FALSE, column names will be generated automatically: X1, X2, X3 etc.
  - If col_names is a character vector, the values will be used as the names of the columns, and the first row of the input will be read into the first row of the output data frame.
  - Missing (NA) column names will generate a warning, and be filled in with dummy names X1, X2 etc. Duplicate column names will generate a warning and be made unique with a numeric suffix.

col_types
- One of NULL, a cols() specification, or a string. See vignette("readr") for more details.
  - If NULL, all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you’ll need to supply the correct types yourself.
  - If a column specification created by cols(), it must contain one column specification for each column. If you only want to read a subset of the columns, use cols_only().
  - Alternatively, you can use a compact string representation where each character represents one column:
    - c = character
    - i = integer
    - n = number
    - d = double
    - l = logical
    - f = factor
    - D = date
    - T = date time
    - t = time
    - ? = guess
    - _ or - = skip
  - By default, reading a file without a column specification will print a message showing what readr guessed they were. To remove this message, use col_types = cols().

locale
- The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use locale() to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

na
- Character vector of strings to interpret as missing values. Set this option to character() to indicate no missing values.
quoted_na Should missing values inside quotes be treated as missing values (the default) or strings.

comment A string used to identify comments. Any text after the comment characters will be silently ignored.

trim_ws Should leading and trailing whitespace be trimmed from each field before parsing it?

skip Number of lines to skip before reading data.

n_max Maximum number of records to read.

guess_max Maximum number of records to use for guessing column types.

progress Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The display is updated every 50,000 values and will only display if estimated reading time is 5 seconds or more. The automatic progress bar can be disabled by setting option readr.show_progress to FALSE.

skip_empty_rows Should blank rows be ignored altogether? i.e. If this option is TRUE then blank rows will not be represented at all. If it is FALSE then they will be represented by NA values in all the columns.

Value

A tibble(). If there are parsing problems, a warning tells you how many, and you can retrieve the details with problems().

Examples

# Input sources -------------------------------------------------------------
# Read from a path
read_csv(readr_example("mtcars.csv"))
read_csv(readr_example("mtcars.csv.zip"))
read_csv(readr_example("mtcars.csv.bz2"))

## Not run:
# Including remote paths
read_csv("https://github.com/tidyverse/readr/raw/master/inst/extdata/mtcars.csv")

## End(Not run)

# Or directly from a string (must contain a newline)
read_csv("x,y\n1,2\n3,4")

# Column types -------------------------------------------------------------
# By default, readr guesses the columns types, looking at the first 1000 rows.
# You can override with a compact specification:
read_csv("x,y\n1,2\n3,4", col_types = "dc")

# Or with a list of column types:
read_csv("x,y\n1,2\n3,4", col_types = list(col_double(), col_character()))

# If there are parsing problems, you get a warning, and can extract
# more details with problems()
y <- read_csv("x\n1\n2\n", col_types = list(col_double()))
y
# File types  
read_csv("a,b\n1.0,2.0")
read_csv2("a;b\n1,0;2,0")
read_tsv("a\tb\n1.0\tt2.0")
read_delim("a|b\n1.0|2.0", delim = ":")

---

**Description**

`read_file()` reads a complete file into a single object: either a character vector of length one, or a raw vector. `write_file()` takes a single string, or a raw vector, and writes it exactly as is. Raw vectors are useful when dealing with binary data, or if you have text data with unknown encoding.

**Usage**

```r
read_file(file, locale = default_locale())
```

**Arguments**

- `file`  
  Either a path to a file, a connection, or literal data (either a single string or a raw vector).
  Files ending in `.gz`, `.bz2`, `.xz`, or `.zip` will be automatically uncompressed.
  Files starting with `http://`, `https://`, `ftp://`, or `ftps://` will be automatically downloaded. Remote .gz files can also be automatically downloaded and decompressed.
  Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path) or be a vector of greater than length 1.
  Using a value of `clipboard()` will read from the system clipboard.

- `locale`  
  The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

- `x`  
  A single string, or a raw vector to write to disk.

- `append`  
  If `FALSE`, will overwrite existing file. If `TRUE`, will append to existing file. In both cases, if the file does not exist a new file is created.

- `path`  
  **Deprecated**

**Value**

`read_file`: A length 1 character vector. `read_lines_raw`: A raw vector.
read_fwf

Examples

```r
read_file(file.path(R.home("doc"), "AUTHORS"))
read_file_raw(file.path(R.home("doc"), "AUTHORS"))

tmp <- tempfile()
x <- format_csv(mtcars[1:6, ])
write_file(x, tmp)
identical(x, read_file(tmp))

read_lines(x)
```

Description

A fixed width file can be a very compact representation of numeric data. It’s also very fast to parse, because every field is in the same place in every line. Unfortunately, it’s painful to parse because you need to describe the length of every field. Readr aims to make it as easy as possible by providing a number of different ways to describe the field structure.

Usage

```r
read_fwf(
  file,
  col_positions,
  col_types = NULL,
  locale = default_locale(),
  na = c("\"", "NA"),
  comment = "",
  trim_ws = TRUE,
  skip = 0,
  n_max = Inf,
  guess_max = min(n_max, 1000),
  progress = show_progress(),
  skip_empty_rows = TRUE
)

fwf_empty(
  file,
  skip = 0,
  skip_empty_rows = FALSE,
  col_names = NULL,
  comment = "",
  n = 100L
)

fwf_widths(widths, col_names = NULL)
fwf_positions(start, end, col_names = NULL)
```
**Arguments**

**file**

Either a path to a file, a connection, or literal data (either a single string or a raw vector).

Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed.

Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.

Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path) or be a vector of greater than length 1.

Using a value of `clipboard()` will read from the system clipboard.

**col_positions**

Column positions, as created by `fwf_empty()`, `fwf_widths()` or `fwf_positions()`.

To read in only selected fields, use `fwf_positions()`. If the width of the last column is variable (a ragged fwf file), supply the last end position as NA.

**col_types**

One of NULL, a `cols()` specification, or a string. See vignette("readr") for more details.

If NULL, all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you’ll need to supply the correct types yourself.

If a column specification created by `cols()`, it must contain one column specification for each column. If you only want to read a subset of the columns, use `cols_only()`.

Alternatively, you can use a compact string representation where each character represents one column:

- `c` = character
- `i` = integer
- `n` = number
- `d` = double
- `l` = logical
- `f` = factor
- `D` = date
- `T` = date time
- `t` = time
- `?` = guess
- `_` or `-` = skip

By default, reading a file without a column specification will print a message showing what `readr` guessed they were. To remove this message, use `col_types = cols()`.

**locale**

The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

**na**

Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
comment  A string used to identify comments. Any text after the comment characters will
be silently ignored.

trim_ws  Should leading and trailing whitespace be trimmed from each field before pars-
ing it?

skip  Number of lines to skip before reading data.

n_max  Maximum number of records to read.

guess_max  Maximum number of records to use for guessing column types.

progress  Display a progress bar? By default it will only display in an interactive session
and not while knitting a document. The display is updated every 50,000 values
and will only display if estimated reading time is 5 seconds or more. The auto-
matic progress bar can be disabled by setting option readr.show_progress to
FALSE.

skip_empty_rows  Should blank rows be ignored altogether? i.e. If this option is TRUE then blank
rows will not be represented at all. If it is FALSE then they will be represented
by NA values in all the columns.

col_names  Either NULL, or a character vector column names.

n  Number of lines the tokenizer will read to determine file structure. By default it
is set to 100.

widths  Width of each field. Use NA as width of last field when reading a ragged fwf
file.

start, end  Starting and ending (inclusive) positions of each field. Use NA as last end field
when reading a ragged fwf file.

...  If the first element is a data frame, then it must have all numeric columns and
either one or two rows. The column names are the variable names. The column
values are the variable widths if a length one vector, and if length two, variable
start and end positions. The elements of ... are used to construct a data frame
with or two rows as above.

See Also

read_table() to read fixed width files where each column is separated by whitespace.

Examples

fwf_sample <- readr_example("fwf-sample.txt")
writeLines(read_lines(fwf_sample))

# You can specify column positions in several ways:
# 1. Guess based on position of empty columns
read_fwf(fwf_sample, fwf_empty(fwf_sample, col_names = c("first", "last", "state", "ssn")))
# 2. A vector of field widths
read_fwf(fwf_sample, fwf_widths(c(20, 10, 12), c("name", "state", "ssn")))
# 3. Paired vectors of start and end positions
read_fwf(fwf_sample, fwf_positions(c(1, 30), c(20, 42), c("name", "ssn")))
# 4. Named arguments with start and end positions
read_fwf(fwf_sample, fwf_cols(name = c(1, 20), ssn = c(30, 42)))
# 5. Named arguments with column widths
read_fwf(fwf_sample, fwf_cols(name = 20, state = 10, ssn = 12))
**read_lines**  

**Read/write lines to/from a file**

**Description**

read_lines() reads up to \( n_{\text{max}} \) lines from a file. New lines are not included in the output. read_lines_raw() produces a list of raw vectors, and is useful for handling data with unknown encoding. write_lines() takes a character vector or list of raw vectors, appending a new line after each entry.

**Usage**

```r
read_lines(
  file,
  skip = 0,
  skip_empty_rows = FALSE,
  n_max = -1L,
  locale = default_locale(),
  na = character(),
  progress = show_progress()
)
```

```r
read_lines_raw(file, skip = 0, n_max = -1L, progress = show_progress())
```

```r
write_lines(
  x,
  file,
  sep = "\n",
  na = "NA",
  append = FALSE,
  path = deprecated()
)
```

**Arguments**

- **file**  
  Either a path to a file, a connection, or literal data (either a single string or a raw vector).

Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.

Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path) or be a vector of greater than length 1.

Using a value of clipboard() will read from the system clipboard.

- **skip**  
  Number of lines to skip before reading data.

- **skip_empty_rows**  
  Should blank rows be ignored altogether? i.e. If this option is TRUE then blank rows will not be represented at all. If it is FALSE then they will be represented by NA values in all the columns.
**n_max** Number of lines to read. If `n_max` is `-1`, all lines in file will be read.

**locale** The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

**na** Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.

**progress** Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The display is updated every 50,000 values and will only display if estimated reading time is 5 seconds or more. The automatic progress bar can be disabled by setting option `readr.show_progress` to `FALSE`.

**x** A character vector or list of raw vectors to write to disk.

**sep** The line separator. Defaults to `\n`, commonly used on POSIX systems like macOS and Linux. For native Windows (CRLF) separators use `\r\n`.

**append** If `FALSE`, will overwrite existing file. If `TRUE`, will append to existing file. In both cases, if the file does not exist a new file is created.

**path** Deprecated

---

### Examples

```r
read_lines(file.path(R.home("doc"), "AUTHORS"), n_max = 10)
read_lines_raw(file.path(R.home("doc"), "AUTHORS"), n_max = 10)

tmp <- tempfile()
write_lines(rownames(mtcars), tmp)
read_lines(tmp)
read_file(tmp) # note trailing \n
write_lines(airquality$Ozone, tmp, na = "-1")
read_lines(tmp)
```

---

**Description**

This is a fairly standard format for log files - it uses both quotes and square brackets for quoting, and there may be literal quotes embedded in a quoted string. The dash, "-", is used for missing values.
read_log

Usage

read_log(
  file,
  col_names = FALSE,
  col_types = NULL,
  skip = 0,
  n_max = Inf,
  progress = show_progress()
)

Arguments

file Either a path to a file, a connection, or literal data (either a single string or a raw vector).
Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed. Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded. Remote gz files can also be automatically downloaded and decompressed.
Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path) or be a vector of greater than length 1.
Using a value of clipboard() will read from the system clipboard.

col_names Either TRUE, FALSE or a character vector of column names.
If TRUE, the first row of the input will be used as the column names, and will not be included in the data frame. If FALSE, column names will be generated automatically: X1, X2, X3 etc.
If col_names is a character vector, the values will be used as the names of the columns, and the first row of the input will be read into the first row of the output data frame.
Missing (NA) column names will generate a warning, and be filled in with dummy names X1, X2 etc. Duplicate column names will generate a warning and be made unique with a numeric suffix.

col_types One of NULL, a cols() specification, or a string. See vignette("readr") for more details.
If NULL, all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you’ll need to supply the correct types yourself.
If a column specification created by cols(), it must contain one column specification for each column. If you only want to read a subset of the columns, use cols_only().
Alternatively, you can use a compact string representation where each character represents one column:
  • c = character
  • i = integer
  • n = number
  • d = double
  • l = logical
  • f = factor
  • D = date
• T = date time
• t = time
• ? = guess
• _ or - = skip

By default, reading a file without a column specification will print a message showing what \texttt{readr} guessed they were. To remove this message, use \texttt{col_types = cols()}.

\begin{itemize}
\item \texttt{skip} \hspace{1cm} Number of lines to skip before reading data.
\item \texttt{n_max} \hspace{1cm} Maximum number of records to read.
\item \texttt{progress} \hspace{1cm} Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The display is updated every 50,000 values and will only display if estimated reading time is 5 seconds or more. The automatic progress bar can be disabled by setting option \texttt{readr.show_progress} to \texttt{FALSE}.
\end{itemize}

\section*{Examples}

\begin{verbatim}
read_log(readr_example("example.log"))
\end{verbatim}

\section*{read_table}

\textit{Read whitespace-separated columns into a tibble}

\section*{Description}

\texttt{read_table()} and \texttt{read_table2()} are designed to read the type of textual data where each column is separated by one (or more) columns of space. 
\texttt{read_table2()} is like \texttt{read_table()}, it allows any number of whitespace characters between columns, and the lines can be of different lengths. 
\texttt{read_table()} is more strict, each line must be the same length, and each field is in the same position in every line. It first finds empty columns and then parses like a fixed width file. 
\texttt{spec_table()} and \texttt{spec_table2()} return the column specifications rather than a data frame.

\section*{Usage}

\begin{verbatim}
read_table(
  file,
  col_names = TRUE,
  col_types = NULL,
  locale = default_locale(),
  na = "NA",
  skip = 0,
  n_max = Inf,
  guess_max = min(n_max, 1000),
  progress = show_progress(),
  comment = "",
  skip_empty_rows = TRUE
)

read_table2(
\end{verbatim}
```r
read_table

file,
  col_names = TRUE,
  col_types = NULL,
  locale = default_locale(),
  na = "NA",
  skip = 0,
  n_max = Inf,
  guess_max = min(n_max, 1000),
  progress = show_progress(),
  comment = "",
  skip_empty_rows = TRUE
)

Arguments

file
  Either a path to a file, a connection, or literal data (either a single string or a raw vector).
  Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed.
  Files starting with http://, https://, ftp://, or ftps:// will be automatically downloaded.
  Remote gz files can also be automatically downloaded and decompressed.
  Literal data is most useful for examples and tests. It must contain at least one new line to be recognised as data (instead of a path) or be a vector of greater than length 1.
  Using a value of `clipboard()` will read from the system clipboard.

col_names
  Either TRUE, FALSE or a character vector of column names.
  If TRUE, the first row of the input will be used as the column names, and will not be included in the data frame. If FALSE, column names will be generated automatically: X1, X2, X3 etc.
  If col_names is a character vector, the values will be used as the names of the columns, and the first row of the input will be read into the first row of the output data frame.
  Missing (NA) column names will generate a warning, and be filled in with dummy names X1, X2 etc. Duplicate column names will generate a warning and be made unique with a numeric suffix.

col_types
  One of NULL, a cols() specification, or a string. See vignette("readr") for more details.
  If NULL, all column types will be imputed from the first 1000 rows on the input.
  This is convenient (and fast), but not robust. If the imputation fails, you’ll need to supply the correct types yourself.
  If a column specification created by cols(), it must contain one column specification for each column. If you only want to read a subset of the columns, use cols_only() .
  Alternatively, you can use a compact string representation where each character represents one column:
    • c = character
    • i = integer
    • n = number
    • d = double
    • l = logical
```
• f = factor  
• D = date  
• T = date time  
• t = time  
• ? = guess  
• _ or - = skip
  By default, reading a file without a column specification will print a message showing what readr guessed they were. To remove this message, use `col_types = cols()`.

locale  The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

na  Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.

skip  Number of lines to skip before reading data.

n_max  Maximum number of records to read.

guess_max  Maximum number of records to use for guessing column types.

progress  Display a progress bar? By default it will only display in an interactive session and not while knitting a document. The display is updated every 50,000 values and will only display if estimated reading time is 5 seconds or more. The automatic progress bar can be disabled by setting option `readr.show_progress` to `FALSE`.

comment  A string used to identify comments. Any text after the comment characters will be silently ignored.

skip_empty_rows  Should blank rows be ignored altogether? i.e. If this option is `TRUE` then blank rows will not be represented at all. If it is `FALSE` then they will be represented by `NA` values in all the columns.

See Also  
`read_fwf()` to read fixed width files where each column is not separated by whitespace. `read_fwf()` is also useful for reading tabular data with non-standard formatting.

Examples

```r
# One corner from http://www.masseyratings.com/cf/compare.htm
massey <- readr_example("massey-rating.txt")
cat(read_file(massey))
read_table(massey)

# Sample of 1978 fuel economy data from
# http://www.fueleconomy.gov/feg/epadata/78data.zip
epa <- readr_example("epa78.txt")
cat(read_file(epa))
read_table(epa, col_names = FALSE)
```
show_progress

Determine progress bars should be shown

Description

Progress bars are shown unless one of the following is TRUE

- The bar is explicitly disabled by setting `options(readr.show_progress = FALSE)`
- The code is run in a non-interactive session (`interactive()` is FALSE).
- The code is run in an RStudio notebook chunk.
- The code is run by knitr / rmarkdown.

Usage

```
show_progress()
```

spec_delim

Generate a column specification

Description

When printed, only the first 20 columns are printed by default. To override, set `options(readr.num_columns)` can be used to modify this (a value of 0 turns off printing).

Usage

```
spec_delim(
  file,
  delim,
  quote = "\\", escape_backslash = FALSE, escape_double = TRUE,
  col_names = TRUE, col_types = NULL, locale = default_locale(),
  na = c("\", "NA"), quoted_na = TRUE, comment = "\\",
  trim_ws = FALSE, skip = 0,
  n_max = 0, guess_max = 1000,
  progress = show_progress(),
  skip_empty_rows = TRUE
)
```

spec_csv

```
spec_csv(
  file,
)
spec_delim

col_names = TRUE,
col_types = NULL,
locale = default_locale(),
na = c("", "NA"),
quoted_na = TRUE,
quote = "\"",
comment = "",
trim_ws = TRUE,
skip = 0,
n_max = 0,
guess_max = 1000,
progress = show_progress(),
skip_empty_rows = TRUE
)

spec_csv2(
  file,
  col_names = TRUE,
col_types = NULL,
locale = default_locale(),
na = c("", "NA"),
quoted_na = TRUE,
quote = "\"",
comment = "",
trim_ws = TRUE,
skip = 0,
n_max = 0,
guess_max = 1000,
progress = show_progress(),
skip_empty_rows = TRUE
)

spec_tsv(
  file,
  col_names = TRUE,
col_types = NULL,
locale = default_locale(),
na = c("", "NA"),
quoted_na = TRUE,
quote = "\"",
comment = "",
trim_ws = TRUE,
skip = 0,
n_max = 0,
guess_max = 1000,
progress = show_progress(),
skip_empty_rows = TRUE
)

spec_table(
  file,
col_names = TRUE,
spec_delim

col_types = NULL,
locale = default_locale(),
na = "NA",
skip = 0,
n_max = 0,
guess_max = 1000,
progress = show_progress(),
comment = "",
skip_empty_rows = TRUE
}

spec_table2(
  file,
  col_names = TRUE,
  col_types = NULL,
  locale = default_locale(),
  na = "NA",
  skip = 0,
  n_max = 0,
  guess_max = 1000,
  progress = show_progress(),
  comment = "",
  skip_empty_rows = TRUE
)

Arguments

file Either a path to a file, a connection, or literal data (either a single string or a raw
vector).

Files ending in .gz, .bz2, .xz, or .zip will be automatically uncompressed.
Files starting with http://, https://, ftp://, or ftps:// will be automatically down-
loaded. Remote gz files can also be automatically downloaded and decom-
pressed.

Literal data is most useful for examples and tests. It must contain at least one
new line to be recognised as data (instead of a path) or be a vector of greater
than length 1.

Using a value of clipboard() will read from the system clipboard.

delim Single character used to separate fields within a record.

quote Single character used to quote strings.

escape_backslash Does the file use backslashes to escape special characters? This is more gen-
eral than escape_double as backslashes can be used to escape the delimiter
character, the quote character, or to add special characters like \n.

escape_double Does the file escape quotes by doubling them? i.e. If this option is TRUE, the
value """" represents a single quote, ".

col_names Either TRUE, FALSE or a character vector of column names.

If TRUE, the first row of the input will be used as the column names, and will
not be included in the data frame. If FALSE, column names will be generated
automatically: X1, X2, X3 etc.
spec_delim

If `col_names` is a character vector, the values will be used as the names of the columns, and the first row of the input will be read into the first row of the output data frame.

Missing (NA) column names will generate a warning, and be filled in with dummy names X1, X2 etc. Duplicate column names will generate a warning and be made unique with a numeric suffix.

**col_types**

One of NULL, a `cols()` specification, or a string. See vignette("readr") for more details.

If NULL, all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you’ll need to supply the correct types yourself.

If a column specification created by `cols()`, it must contain one column specification for each column. If you only want to read a subset of the columns, use `cols_only()`.

Alternatively, you can use a compact string representation where each character represents one column:

- c = character
- i = integer
- n = number
- d = double
- l = logical
- f = factor
- D = date
- T = date time
- t = time
- ? = guess
- _ or - = skip

By default, reading a file without a column specification will print a message showing what `readr` guessed they were. To remove this message, use `col_types = cols()`.

**locale**

The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

**na**

Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.

**quoted_na**

Should missing values inside quotes be treated as missing values (the default) or strings.

**comment**

A string used to identify comments. Any text after the comment characters will be silently ignored.

**trim_ws**

Should leading and trailing whitespace be trimmed from each field before parsing it?

**skip**

Number of lines to skip before reading data.

**n_max**

Maximum number of records to read.

**guess_max**

Maximum number of records to use for guessing column types.
**type_convert**

### Description

This is useful if you need to do some manual munging - you can read the columns in as character, clean it up with (e.g.) regular expressions and then let readr take another stab at parsing it. The name is a homage to the base `utils::type.convert()`.

### Usage

```r
type_convert(
  df,
  col_types = NULL,
  na = c("", "NA"),
  trim_ws = TRUE,
  locale = default_locale()
)
```
Arguments

- **df**: A data frame.
- **col_types**: One of `NULL`, a `cols()` specification, or a string. See `vignette("readr")` for more details.
  
  If `NULL`, column types will be imputed using all rows.
- **na**: Character vector of strings to interpret as missing values. Set this option to `character()` to indicate no missing values.
- **trim_ws**: Should leading and trailing whitespace be trimmed from each field before parsing it?
- **locale**: The locale controls defaults that vary from place to place. The default locale is US-centric (like R), but you can use `locale()` to create your own locale that controls things like the default time zone, encoding, decimal mark, big mark, and day/month names.

Note

- `type_convert()` removes a 'spec' attribute, because it likely modifies the column data types. (see `spec()` for more information about column specifications).

Examples

```r
df <- data.frame(
  x = as.character(runif(10)),
  y = as.character(sample(10)),
  stringsAsFactors = FALSE
)
str(df)
str(type_convert(df))

df <- data.frame(x = c("NA", "10"), stringsAsFactors = FALSE)
str(type_convert(df))

# Type convert can be used to infer types from an entire dataset

# first read the data as character
data <- read_csv(readr_example("mtcars.csv"),
  col_types = cols(.default = col_character()))
str(data)
# Then convert it with type_convert
type_convert(data)
```

write_delim

Write a data frame to a delimited file

Description

The `write_*()` family of functions are an improvement to analogous function such as `write.csv()` because they are approximately twice as fast. Unlike `write.csv()`, these functions do not include row names as a column in the written file. A generic function, `output_column()`, is applied to each variable to coerce columns to suitable output.
write_delim

Usage

write_delim(
  x,
  file,
  delim = " ",
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote_escape = "double",
  eol = "\n",
  path = deprecated()
)

write_csv(
  x,
  file,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote_escape = "double",
  eol = "\n",
  path = deprecated()
)

write_csv2(
  x,
  file,
  na = "NA",
  append = FALSE,
  col_names = !append,
  quote_escape = "double",
  eol = "\n",
  path = deprecated()
)

write_excel_csv(
  x,
  file,
  na = "NA",
  append = FALSE,
  col_names = !append,
  delim = ",",
  quote_escape = "double",
  eol = "\n",
  path = deprecated()
)

write_excel_csv2(
  x,
  file,
  na = "NA",
  append = FALSE,
col_names = !append,
delim = ";",
quote_escape = "double",
eol = "\n",
path = deprecated()
)

write_tsv(
x,
file,
na = "NA",
append = FALSE,
col_names = !append,
quote_escape = "double",
eol = "\n",
path = deprecated()
)

Arguments

x  A data frame or tibble to write to disk.
file File or connection to write to.
delim Delimiter used to separate values. Defaults to "," for write_delim(), ",," for write_excel_csv() and ";" for write_excel_csv2(). Must be a single character.
na String used for missing values. Defaults to NA. Missing values will never be quoted; strings with the same value as na will always be quoted.
append If FALSE, will overwrite existing file. If TRUE, will append to existing file. In both cases, if the file does not exist a new file is created.
col_names If FALSE, column names will not be included at the top of the file. If TRUE, column names will be included. If not specified, col_names will take the opposite value given to append.
quote_escape The type of escaping to use for quoted values, one of "double", "backslash" or "none". You can also use FALSE, which is equivalent to "none". The default is "double", which is expected format for Excel.
eol The end of line character to use. Most commonly either "\n" for Unix style newlines, or "\r\n" for Windows style newlines.
path Deprecated

Value

write_*() returns the input x invisibly.

Output

Factors are coerced to character. Doubles are formatted to a decimal string using the grisu3 algorithm. POSIXct values are formatted as ISO8601 with a UTC timezone Note: POSIXct objects in local or non-UTC timezones will be converted to UTC time before writing.

All columns are encoded as UTF-8. write_excel_csv() and write_excel_csv2() also include a UTF-8 Byte order mark which indicates to Excel the csv is UTF-8 encoded.
write_delim

write_csv2() and write_csv2 were created to allow users with different locale settings to save .csv files using their default settings (e.g. ; as the column separator and , as the decimal separator). This is common in some European countries.

Values are only quoted if they contain a comma, quote or newline.

The write_*() functions will automatically compress outputs if an appropriate extension is given. Three extensions are currently supported: .gz for gzip compression, .bz2 for bzip2 compression and .xz for lzma compression. See the examples for more information.

References


Examples

data(storms, package = "dplyr")
# If only a file name is specified, write_()* will write
# the file to the current working directory.
write_csv(storms, "storms.csv")
write_tsv(storms, "storms.tsv")

# If you add an extension to the file name, write_()* will
# automatically compress the output.
write_tsv(storms, "storms.tsv.gz")
write_tsv(storms, "storms.tsv.bz2")
write_tsv(storms, "storms.tsv.xz")
Index

* parsers
  col_skip, 5
  cols, 3
  cols_condense, 4
  parse_atomic, 17
  parse_datetime, 19
  parse_factor, 22
  parse_guess, 23
  parse_number, 24

clipboard, 3
clipboard(), 6, 12, 14, 17, 28, 31, 33, 35, 37, 39, 43

col_character (parse_atomic), 17
col_date (parse_datetime), 19
col_datetime (parse_datetime), 19
col_double (parse_atomic), 17
col_factor (parse_factor), 22
col_guess (parse_guess), 23
col_logical (parse_atomic), 17
col_number (parse_number), 24
col_skip, 4, 5, 5, 18, 20, 22, 24, 25
col_time (parse_datetime), 19
cols, 3, 5, 18, 20, 22, 24, 25
cols(), 29, 33, 37, 39, 44, 46
cols_condense, 4, 4, 5, 18, 20, 22, 24, 25
cols_only(cols), 3
cols_only(), 5, 29, 33, 37, 39, 44
count_fields, 5

data(), 26
datasource(), 9
date_names, 6
date_names(), 10
date_names_lang (date_names), 6
date_names_lang(), 10
date_names_langs (date_names), 6
default_locale (locale), 9

factor(), 22
format_csv (format_delim), 7
format_csv2 (format_delim), 7
format_delim, 7

format_tsv (format_delim), 7
fwf_cols (read_fwf), 32
fwf_empty (read_fwf), 32
fwf_empty(), 14, 33
fwf_positions (read_fwf), 32
fwf_positions(), 14, 33
fwf_widths (read_fwf), 32
fwf_widths(), 14, 33
guess_encoding, 9
guess_parser (parse_guess), 23

locale, 9
locale(), 12, 14, 17–19, 22–24, 29, 31, 33, 36, 40, 44, 46

melt_csv (melt_delim), 11
melt_csv2 (melt_delim), 11
melt_delim, 11
melt_fwf, 14
melt_fwf(), 17
melt_table, 16
melt_table(), 15
melt_table2 (melt_table), 16
melt_tsv (melt_delim), 11
OlsonNames(), 10

parse_atomic, 17
parse_character (parse_atomic), 17
parse_date (parse_datetime), 19
parse_datetime, 4, 5, 18, 19, 22, 24, 25
parse_double (parse_atomic), 17
parse_factor, 4, 5, 18, 20, 22, 24, 25
parse_guess, 4, 5, 18, 20, 22, 23, 25
parse_integer (parse_atomic), 17
parse_logical, 4, 5, 20, 22, 24, 25
parse_logical (parse_atomic), 17
parse_number, 4, 5, 18, 20, 22, 24, 24
parse_time (parse_datetime), 19
parse_vector, 4, 5, 18, 20, 22, 24, 25
POSIXct(), 20
problems, 25
problems(), 13, 30
INDEX

read.table(), 38
read_builtin, 26
read_csv (read_delim), 27
read_csv2 (read_delim), 27
read_delim, 27
read_delim(), 3, 13
read_file, 31
read_file_raw (read_file), 31
read_fwf, 32
read_fwf(), 15, 40
read_lines, 35
read_lines_raw (read_lines), 35
read_log, 36
read_table, 38
read_table(), 17, 34
read_table2 (read_table), 38
read_tsv (read_delim), 27
readr_example, 26

show_progress, 41
spec (cols_condense), 4
spec(), 46
spec_csv (spec_delim), 41
spec_csv2 (spec_delim), 41
spec_delim, 41
spec_table (spec_delim), 41
spec_table2 (spec_delim), 41
spec_tsv (spec_delim), 41
stop_for_problems (problems), 25
stringi::stri_enc_detect(), 9
strptime(), 19, 20

tibble(), 13, 30
tokenizer_csv(), 6
tokenizer_fwf(), 6
type_convert, 45

utils::type.convert(), 45

write.csv(), 46
write.csv (write_delim), 46
write.csv(), 7
write.csv2 (write_delim), 46
write_delim, 46
write_excel_csv (write_delim), 46
write_excel_csv2 (write_delim), 46
write_file (read_file), 31
write_lines (read_lines), 35
write_tsv (write_delim), 46